확률 및 통계 Assignment3

제목: Card counting

과제 기간: 2022년 5월 11일 (수)

~

2022년 5월 24일 (화)

학 과: 컴퓨터정보공학부

담당교수: 심동규 교수님

수업시간: 월, 수 5,6교시

학 번: 2018202065

성 명: 박 철 준

- 소개

1. 제목

Card counting

2. 목적 및 요구 사항

가. 블랙잭 카드 게임을 구현하고 카드카운팅을 통하여 승률을 올려야 한다.

나. Dealer, player 1, player 2가 같이 게임을 진행한다.

다. 플레이어들은 딜러의 카드 한 장을 볼 수 없다

라. 처음 두 장의 카드는 Dealer → player 1 → player 2 순서로 받는다.

마. player 1은 카드 카운팅을 통한 확률 기반으로 Betting과 Hit, Stand를 결정하여야 한다.

바. player 2는 딜러와 마찬가지로 17이상이면 Stand, 미만이면 Hit을 한다.

사. 게임을 n번 진행 후, player 1과 player 2의 승률을 계산 후 비교한다.

아. 카드는 3 deck(156)으로 진행하며, 카드의 80% 이상 사용한다면 다음 게임을 시작할 때모든 카드(사용한 카드 포함)를 다시 섞어준다.

자. 초기 자금은 100000원이다.

차. Player 1과 Player 2는 각각 게임 시작 시 100~1000원, 500원의 금액을 베팅한다.

카. Shuffle() : 카드를 섞어주는 기능을 한다. 6 deck의 80% 이상 사용하면, 다음 게임을 시작할 때, 모든 카드(사용한 카드 포함)를 다시 섞어준다.

타. HitStatus() : 카드를 Hit할지, Stand할지 정해주는 기능을 한다. Cardcounting을 고려한 HitStatus(), dealer와 simple player의 HitStatus()를 고려해서 작성한다.

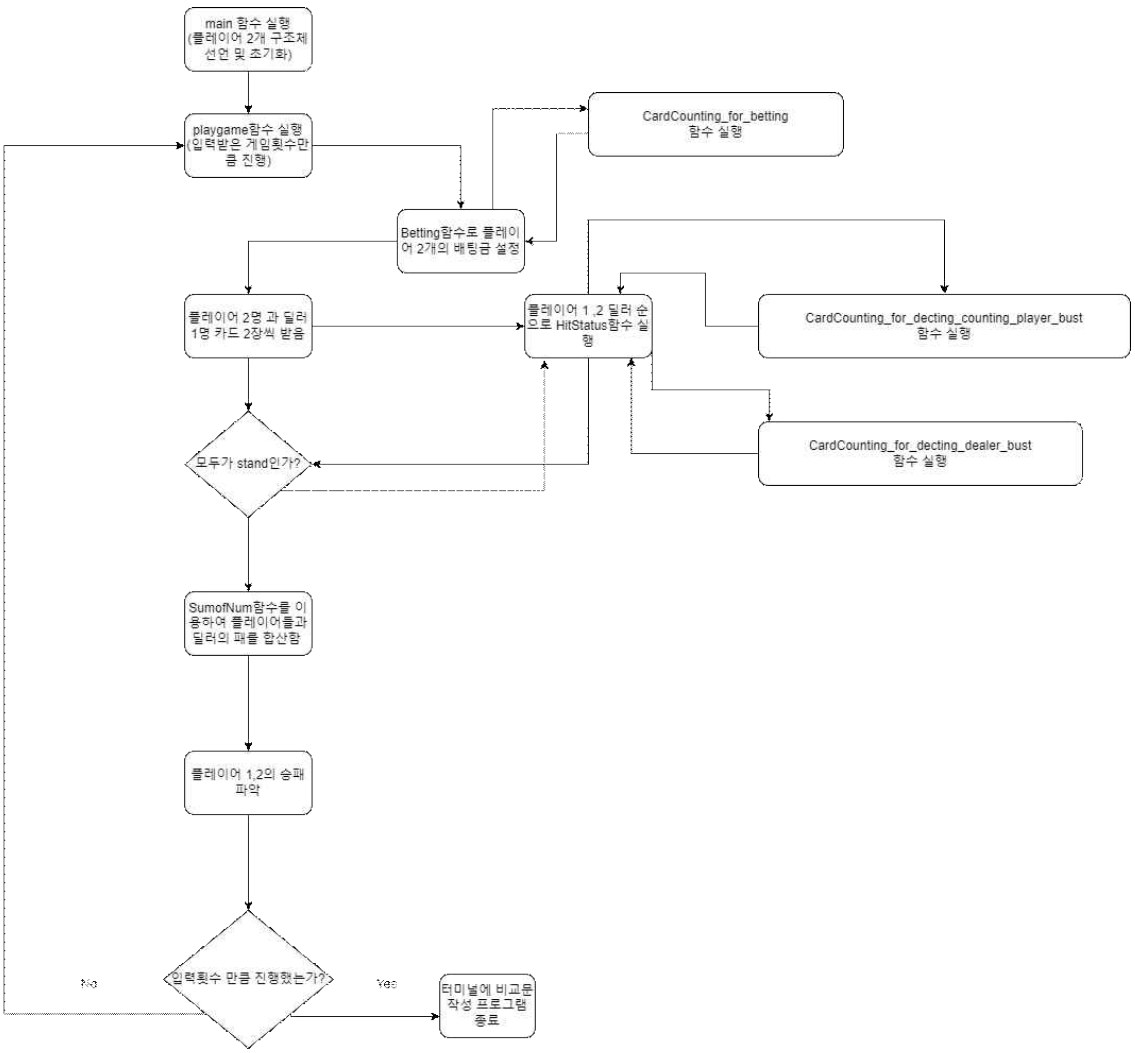
파. SumofNum() : 현재 플레이어의 카드 숫자 합이 얼마인지 계산하는 기능을 한다.

하. - 이외에 필요한 기능을 자유롭게 작성한다.

● 설계 및 구현

- 접근 방법

-SW 순서도
먼저 프로그램을 설계하기 위해 SW 순서도를 작성하였고 다음과 같다.



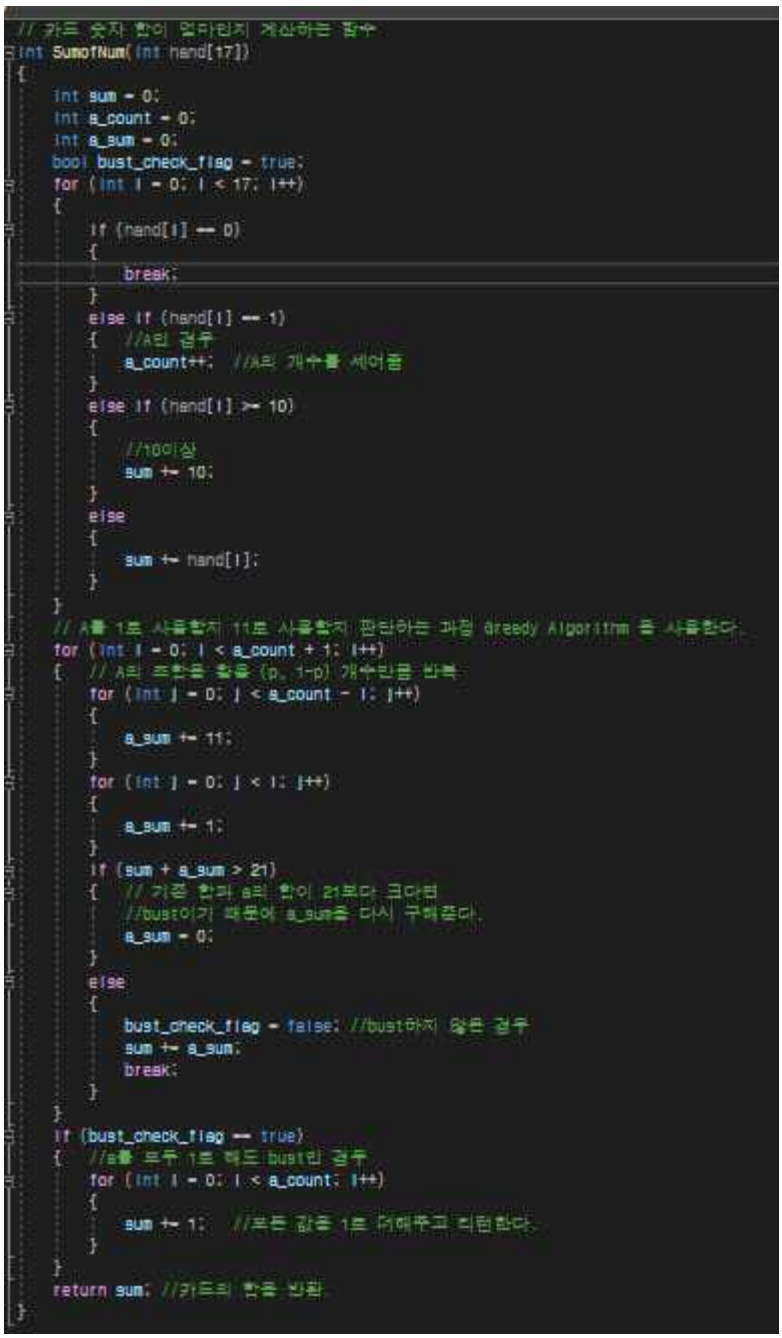
-각 함수 설명
프로그램을 구현하는데 필요한 함수를 흐름도를 참고하여 작성하였고 다음과 같다.

Shuffle() 함수

```
void Shuffle(vector<int>* card_deck)
{// 카드 섞어 주는 함수
    if (!card_deck->empty())    //덱이 비어있지 않다면
        card_deck->clear(); // 남은 카드를 모두 삭제해야한다.

    for (int i = 0; i < 12; i++) {
        card_deck->push_back(1);    //A부터
        card_deck->push_back(2);
        card_deck->push_back(3);
        card_deck->push_back(4);
        card_deck->push_back(5);
        card_deck->push_back(6);
        card_deck->push_back(7);
        card_deck->push_back(8);
        card_deck->push_back(9);
        card_deck->push_back(10);
        card_deck->push_back(11);
        card_deck->push_back(12);
        card_deck->push_back(13);   //K까지
    }
    std::random_shuffle(card_deck->begin(), card_deck->end());
}
```

벡터를 사용하여 푸시하여 값을 넣어주고 이것을 셔플하여 사용할 수 있다.

SumofNum()함수



```
// 카드 숫자 합이 얼마인지 계산하는 함수
int SumofNum(int hand[17])
{
    int sum = 0;
    int a_count = 0;
    int a_sum = 0;
    bool bust_check_flag = true;
    for (int i = 0; i < 17; i++)
    {
        if (hand[i] == 0)
        {
            break;
        }
        else if (hand[i] == 1)
        {   //A인 경우
            a_count++;  //A의 개수를 세어줌
        }
        else if (hand[i] >= 10)
        {
            //10이상
            sum += 10;
        }
        else
        {
            sum += hand[i];
        }
    }
    // A를 1로 사용할지 11로 사용할지 판단하는 과정 Greedy Algorithm 을 사용한다.
    for (int i = 0; i < a_count + 1; i++)
    {   // A의 조합을 활용 (p, 1-p) 개수만큼 반복
        for (int j = 0; j < a_count - i; j++)
        {
            a_sum += 11;
        }
        for (int j = 0; j < i; j++)
        {
            a_sum += 1;
        }
        if (sum + a_sum > 21)
        {   // 기존 합과 a의 합이 21보다 크다면
            //bust이기 때문에 a_sum을 다시 구해준다.
            a_sum = 0;
        }
        else
        {
            bust_check_flag = false; //bust하지 않은 경우
            sum += a_sum;
            break;
        }
    }
    if (bust_check_flag == true)
    {  //a를 모두 1로 해도 bust인 경우
        for (int i = 0; i < a_count; i++)
        {
            sum += 1;   //모든 값을 1로 더해주고 리턴한다.
        }
    }
    return sum; //카드의 합을 반환
}
```

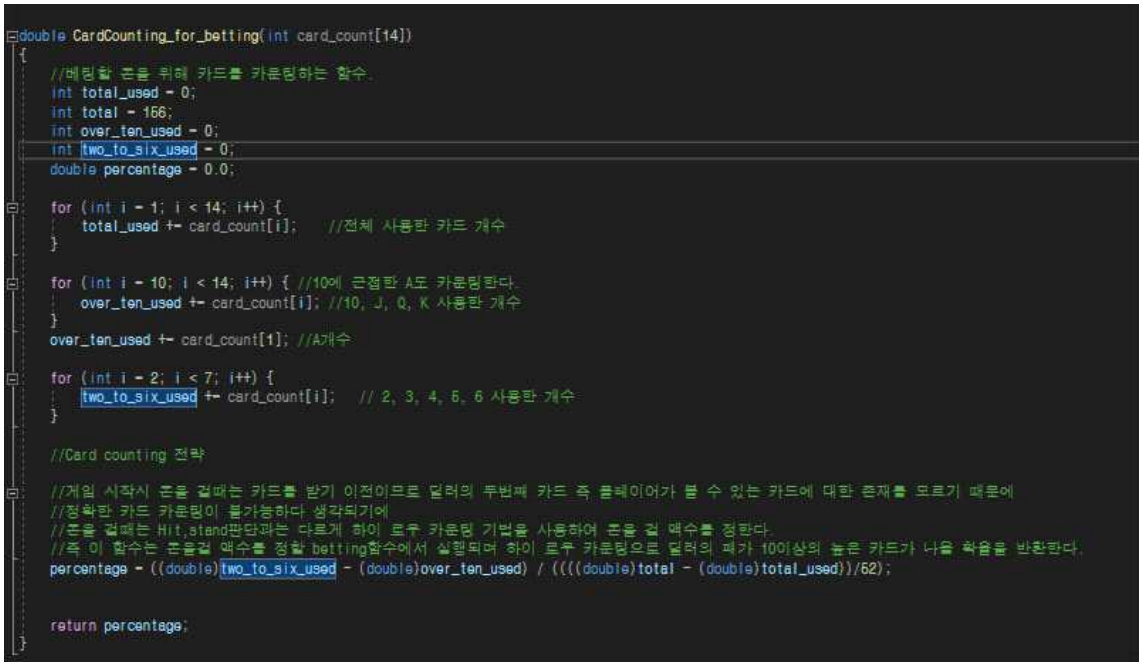
먼저 a의 사용 개수를 저장하고 두 번째 for문으로 그리디 알고리즘을 통해 A의 조합에 대해 P 개수 = 1로 하는 경우 개수, P-1 개수 = 11로 하는 경우의 개수를 반복적으로 확인하여 21이하이고 가장 작은 값을 찾아 낸다.

SumofNum_for_dealer()함수

```
int SumofNum_for_dealer(int hand[17])
{
    int sum = 0;
    int a_count = 0;
    int a_sum = 0;
    bool bust_check_flag = true;
    for (int i = 0; i < 17; i++)
    {
        if (hand[i] == 0)
        {
            break;
        }
        else if (hand[i] == 1)
        {   //A인 경우
            sum += 11;  //A의 개수를 세어줌
        }
        else if (hand[i] >= 10)
        {
            //10이상
            sum += 10;
        }
        else
        {
            sum += hand[i];
        }
    }

    return sum; //카드의 합을 반환
}
```

딜러는 A를 11로 밖에 못 쓰므로 예외 처리를 두어 각 카드를 다 더하는 형식이다.

CardCounting_for_betting()함수



```
double CardCounting_for_betting(int card_count[14])
{
    //베팅할 돈을 위해 카드를 카운팅하는 함수.
    int total_used = 0;
    int total = 166;
    int over_ten_used = 0;
    int two_to_six_used = 0;
    double percentage = 0.0;

    for (int i = 1; i < 14; i++) {
        total_used += card_count[i];    //전체 사용한 카드 개수
    }

    for (int i = 10; i < 14; i++) { //10에 근접한 A도 카운팅한다.
        over_ten_used += card_count[i]; //10, J, Q, K 사용한 개수
    }
    over_ten_used += card_count[1]; //A개수

    for (int i = 2; i < 7; i++) {
        two_to_six_used += card_count[i];   // 2, 3, 4, 5, 6 사용한 개수
    }

    //Card counting 전략

    //게임 시작시 돈을 걸때는 카드를 받기 이전이므로 딜러의 두번째 카드 즉 플레이어가 볼 수 있는 카드에 대한 존재를 모르기 때문에
    //정확한 카드 카운팅이 불가능하다 생각되기에
    //돈을 걸때는 Hit,stand판단과는 다르게 하이 로우 카운팅 기법을 사용하여 돈을 걸 액수를 정한다.
    //즉 이 함수는 돈을걸 액수를 정할 betting함수에서 실행되며 하이 로우 카운팅으로 딜러의 패가 10이상의 높은 카드가 나올 확률을 반환한다.
    percentage = ((double)two_to_six_used - (double)over_ten_used) / ((((double)total - (double)total_used))/52);


    return percentage;
}
```

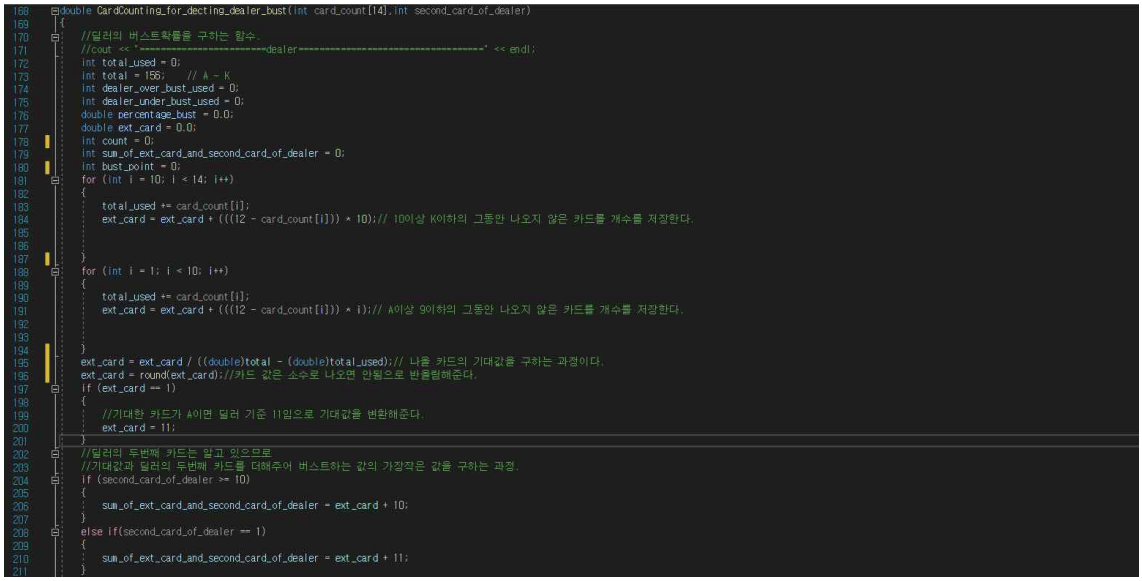
게임 시작시 돈을 걸때는 카드를 받기 이전이므로 딜러의 두번째 카드 즉 플레이어가 볼 수 있는 카드에 대한 존재를 모르기 때문에 정확한 카드 카운팅이 불가능하다 생각되기에 돈을 걸때는 Hit,stand판단과는 다르게 하이 로우 카운팅 기법을 사용하여 돈을 걸 액수를 정한다. 즉 이 함수는 돈을 걸 액수를 정할 betting함수에서 실행되며 하이 로우 카운팅으로 딜러의 패가 10이상의 높은 카드가 나올 확률을 반환한다.

Betting() 함수

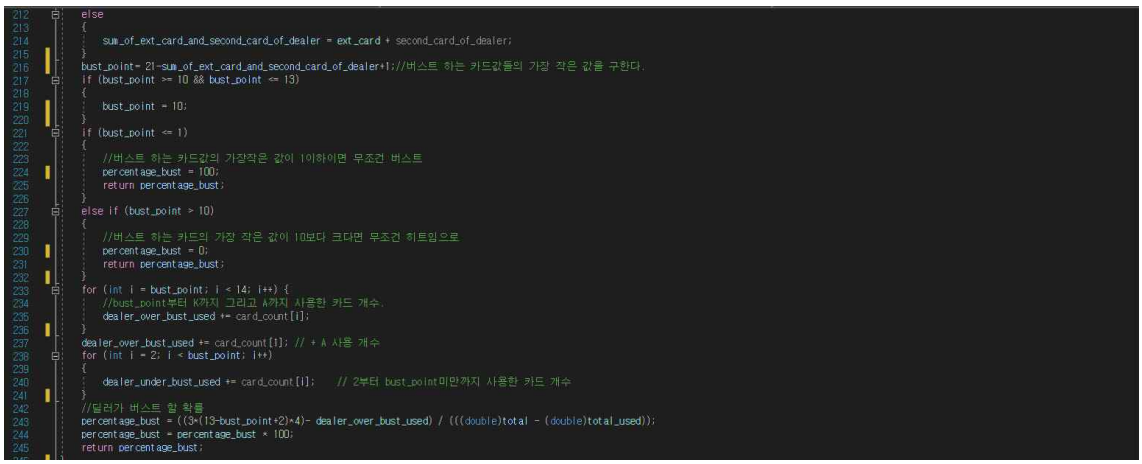
```
// 베팅 금액 설정 함수
double Betting(int card_count[14], bool counting_flag)
{
    double hi_lo_counting = 0.0;
    if (counting_flag) {
        //카운팅 플레이어는 확률에 따라 베팅한다.

        hi_lo_counting = CardCounting_for_betting(card_count);
        if ((int)hi_lo_counting<=0)
        {
            //하이 로우 카운팅 값이 0또는 그 이하일 때 딜러의 버스트 확률이 가장 작다 그러므로 돈을 가장 적게 건다.
            return 100;
        }
        else if ((int)hi_lo_counting > 0 && (int)hi_lo_counting <= 10)
        {
            //하이 로우 카운팅 값이 0보다 크고 10이하일 때 하이 로우 카운팅 값에 따라 돈을 건다.
            return (int)hi_lo_counting * 100;
        }
        else
        {
            //딜러의 버스트 확률이 매우 크므로
            return 1000;
        }

    }
    else
    {
        //simple player는 600원씩 베팅한다.
        return 600;
    }
}
```

CardCounting_for_betting()의 반환값인 하이 로우 카운팅 값을 10 이상일 때 딜러가 버스트 할 확률이 가장 높다고 보아 카운팅값에 1000원을 반환하고 카운팅 값이 0보다 크고 10보다 작을 때는 카운팅 값이 커질 때마다 딜러의 버스트 확률이 올라간다고 보아 카운팅값에 100원을 곱해서 반환하고 카운팅 값이 0포함 작고 음수일 때 딜러의 버스트 확률 보다 카운팅 플레이어의 버스트 확률이 올라간다고 보아 100원을 반환한다.

CardCounting_for_decting_dealer_bust() 함수



```
double CardCounting_for_decting_dealer_bust(int card_count[14],int second_card_of_dealer)
{
    //딜러의 버스트확률을 구하는 함수.
    //cout << "=======================dealer==================================" << endl;
    int total_used = 0;
    int total = 156;    // A ~ K
    int dealer_over_bust_used = 0;
    int dealer_under_bust_used = 0;
    double percentage_bust = 0.0;
    double ext_card = 0.0;
    int count = 0;
    int sum_of_ext_card_and_second_card_of_dealer = 0;
    int bust_point = 0;
    for (int i = 10; i < 14; i++)
    {
        total_used += card_count[i];
        ext_card = ext_card + (((12 - card_count[i])) * 10);// 10이상 K이하의 그동안 나오지 않은 카드를 개수를 저장한다.


    }
    for (int i = 1; i < 10; i++)
    {
        total_used += card_count[i];
        ext_card = ext_card + (((12 - card_count[i])) * i);// A이상 9이하의 그동안 나오지 않은 카드를 개수를 저장한다.


    }
    ext_card = ext_card / ((double)total - (double)total_used);// 나올 카드의 기대값을 구하는 과정이다.
    ext_card = round(ext_card);//카드 값은 소수로 나오면 안됨으로 반올림해준다.
    if (ext_card == 1)
    {
        //기대한 카드가 A이면 딜러 기준 11임으로 기대값을 변환해준다.
        ext_card = 11;
    }
    //딜러의 두번째 카드는 알고 있으므로
    //기대값과 딜러의 두번째 카드를 더해주어 버스트하는 값의 가장작은 값을 구하는 과정.
    if (second_card_of_dealer >= 10)
    {
        sum_of_ext_card_and_second_card_of_dealer = ext_card + 10;
    }
    else if(second_card_of_dealer == 1)
    {
        sum_of_ext_card_and_second_card_of_dealer = ext_card + 11;
    }
```

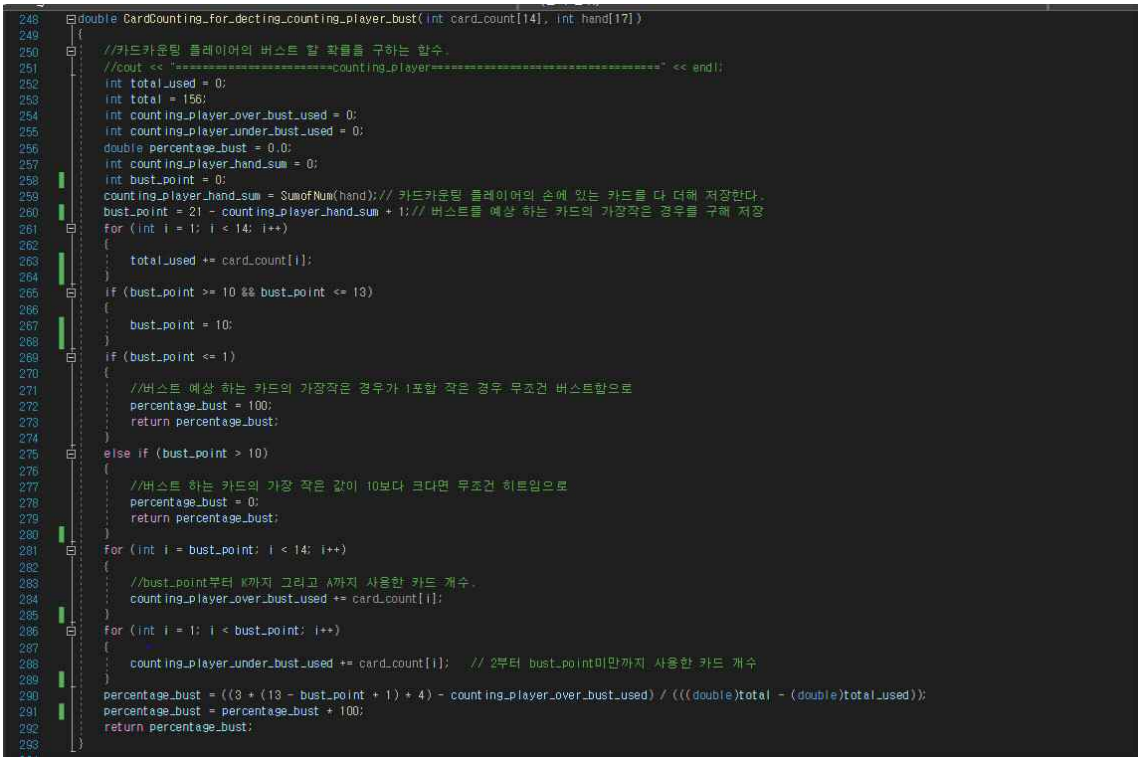


```
    else
    {
        sum_of_ext_card_and_second_card_of_dealer = ext_card + second_card_of_dealer;
    }
    bust_point= 21-sum_of_ext_card_and_second_card_of_dealer+1;//버스트 하는 카드값들의 가장 작은 값을 구한다.
    if (bust_point >= 10 && bust_point <= 13)
    {
        bust_point = 10;
    }
    if (bust_point <= 1)
    {
        //버스트 하는 카드값의 가장작은 값이 1이하이면 무조건 버스트
        percentage_bust = 100;
        return percentage_bust;
    }
    else if (bust_point > 10)
    {
        //버스트 하는 카드의 가장 작은 값이 10보다 크다면 무조건 히트임으로
        percentage_bust = 0;
        return percentage_bust;
    }
    for (int i = bust_point; i < 14; i++) {
        //bust_point부터 K까지 그리고 A까지 사용한 카드 개수.
        dealer_over_bust_used += card_count[i];
    }
    dealer_over_bust_used += card_count[1]; // + A 사용 개수
    for (int i = 2; i < bust_point; i++)
    {
        dealer_under_bust_used += card_count[i];    // 2부터 bust_point미만까지 사용한 카드 개수
    }
    //딜러가 버스트 할 확률
    percentage_bust = ((3*(13-bust_point+2)*4)- dealer_over_bust_used) / (((double)total - (double)total_used));
    percentage_bust = percentage_bust * 100;
    return percentage_bust;
```

이번 프로그램의 가장 핵심의 역할을 하는 함수로 딜러의 버스트 확률을 가장 정확히 계산하여 반환한다.

먼저 함수를 보면 지금까지 나오지 않은 모든 카드를 저장하여 기대값을 구하는데 이는 각 숫자 카드의 문양은 상관없이 남은 빈도수를 가지고 파악한다. 즉 사용하지 않은 카드의 나타내는 숫자와 그에 대한 빈도수를 곱하고 이러한 것을 카드 각각으로 하여 나온 결과를 더해주고 종합적으로 남아 있는 카드 숫자로 나누어 주어 기댓값을 구해주었고 이 값을 딜러의 첫 번째 카드로 예상하고 알고 있는 딜러의 첫 번째 카드와 더해준다. 이후 이 값으로 인해 버스트 되는 세 번째 카드의 가장 작은 숫자를 구한다. 이것이 bust_point로 정의하였고 이 bust_point에서 A(11)까지의 남아 있는 계수를 파악하여 토탈로 남아있는 카드로 나누어 주면 버스트 되는 카드가 나올 확률이 구해지는 데 이 값을 HitStatus함수에서 가져와서 Hit or stand를 결정한다.

CardCounting_for_decting_counting_player_bust() 함수

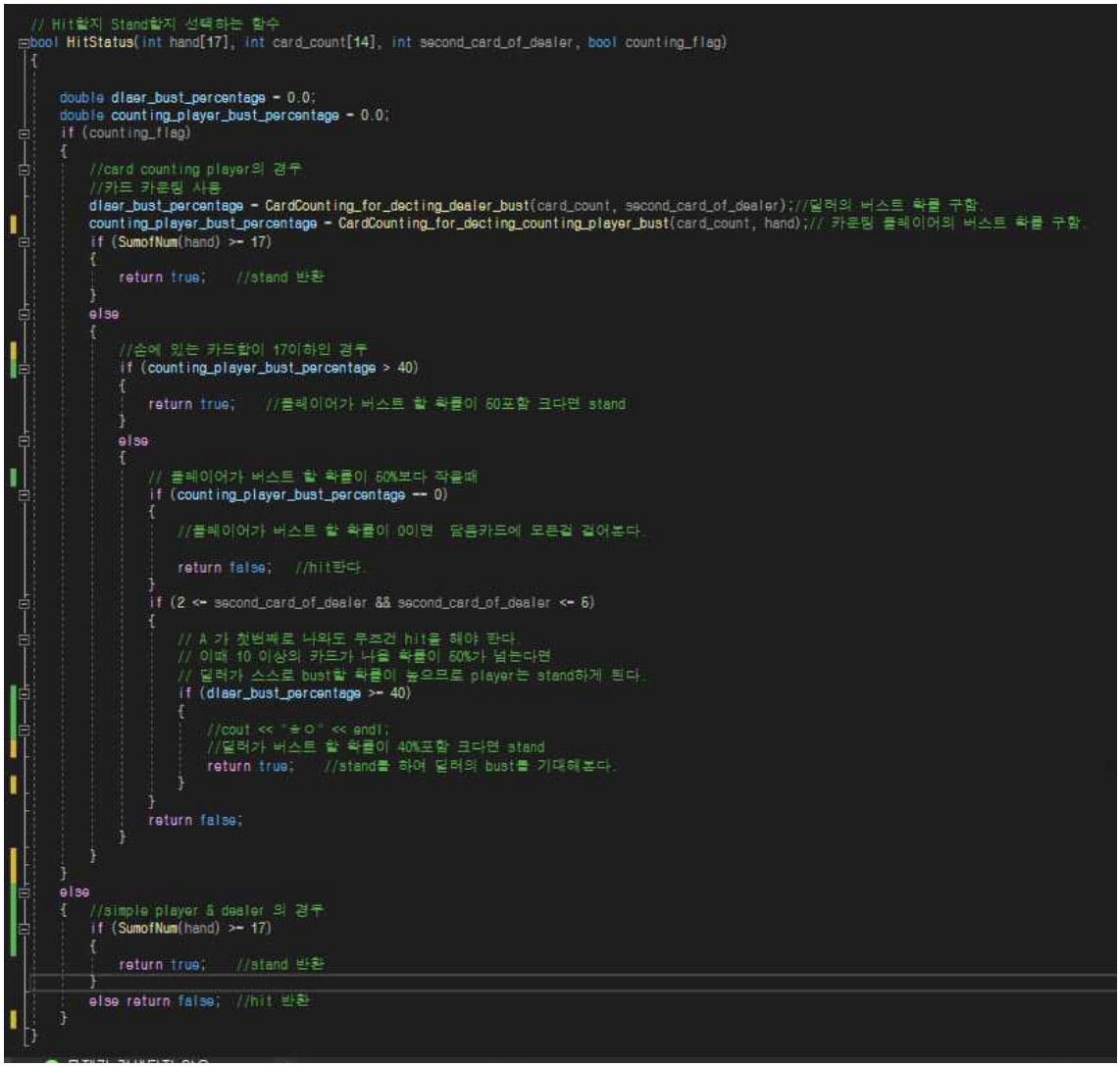


```
double CardCounting_for_decting_counting_player_bust(int card_count[14], int hand[17])
{
    //카드카운팅 플레이어의 버스트 할 확률을 구하는 함수.
    //cout << "=====================counting_player==================================" << endl;
    int total_used = 0;
    int total = 156;
    int counting_player_over_bust_used = 0;
    int counting_player_under_bust_used = 0;
    double percentage_bust = 0.0;
    int counting_player_hand_sum = 0;
    int bust_point = 0;
    counting_player_hand_sum = SumofNum(hand);// 카드카운팅 플레이어의 손에 있는 카드를 다 더해 저장한다.
    bust_point = 21 - counting_player_hand_sum + 1;// 버스트를 예상 하는 카드의 가장작은 경우를 구해 저장
    for (int i = 1; i < 14; i++)
    {
        total_used += card_count[i];
    }
    if (bust_point >= 10 && bust_point <= 13)
    {
        bust_point = 10;
    }
    if (bust_point <= 1)
    {
        //버스트 예상 하는 카드의 가장작은 경우가 1포함 작은 경우 무조건 버스트함으로
        percentage_bust = 100;
        return percentage_bust;
    }
    else if (bust_point > 10)
    {
        //버스트 하는 카드의 가장 작은 값이 10보다 크다면 무조건 히트임으로
        percentage_bust = 0;
        return percentage_bust;
    }
    for (int i = bust_point; i < 14; i++)
    {
        //bust_point부터 K까지 그리고 A까지 사용한 카드 개수.
        counting_player_over_bust_used += card_count[i];
    }
    for (int i = 1; i < bust_point; i++)
    {
        counting_player_under_bust_used += card_count[i];   // 2부터 bust_point미만까지 사용한 카드 개수
    }
    percentage_bust = ((3 + (13 - bust_point + 1) + 4) - counting_player_over_bust_used) / (((double)total - (double)total_used));
    percentage_bust = percentage_bust * 100;
    return percentage_bust;
}
```

이번 프로그램의 가장 핵심의 역할을 하는 두번째 함수로 카운팅 플레이어의 버스트 확률을 가장 정확히 계산하여 반환한다.

먼저 함수를 보면 카운팅 플레이어가 가지고 있는 카드를 그리디 알고리즘으로 가장 최저의 값을 찾아주는 SumofNum함수를 통해 카드의 합산을 구하고 이후 이 값으로 인해 버스트 되는 세 번째 카드의 가장 작은 숫자를 구한다. 이것이 bust_point로 정의하였고 이 bust_point에서 K(10)까지의 남아 있는 계수를 파악하여 토탈로 남아있는 카드로 나누어 주면 버스트 되는 카드가 나올 확률이 구해지는 데 이 값을 HitStatus함수에서 가져와서 Hit or stand를 결정한다.

HitStatus()함수



```
// Hit할지 Stand할지 선택하는 함수
bool HitStatus(int hand[17], int card_count[14], int second_card_of_dealer, bool counting_flag)
{

    double dlaer_bust_percentage = 0.0;
    double counting_player_bust_percentage = 0.0;
    if (counting_flag)
    {
        //card counting player의 경우
        //카드 카운팅 사용
        dlaer_bust_percentage = CardCounting_for_decting_dealer_bust(card_count, second_card_of_dealer);//딜러의 버스트 확률 구함.
        counting_player_bust_percentage = CardCounting_for_decting_counting_player_bust(card_count, hand);// 카운팅 플레이어의 버스트 확률 구함.
        if (SumofNum(hand) >= 17)
        {
            return true;    //stand 반환
        }
        else
        {
            //손에 있는 카드합이 17이하인 경우
            if (counting_player_bust_percentage > 40)
            {
                return true;    //플레이어가 버스트 할 확률이 60포함 크다면 stand
            }
            else
            {
                // 플레이어가 버스트 할 확률이 60%보다 작을때
                if (counting_player_bust_percentage == 0)
                {
                    //플레이어가 버스트 할 확률이 0이면  담음카드에 모른걸 걸어본다.

                    return false;   //hit한다.
                }
                if (2 <= second_card_of_dealer && second_card_of_dealer <= 5)
                {
                    // A 가 첫번째로 나와도 무조건 hit을 해야 한다.
                    // 이때 10 이상의 카드가 나올 확률이 60%가 넘는다면
                    // 딜러가 스스로 bust할 확률이 높으므로 player는 stand하게 된다.
                    if (dlaer_bust_percentage >= 40)
                    {
                        //cout << "ㅎㅇ" << endl;
                        //딜러가 버스트 할 확률이 40%포함 크다면 stand
                        return true;    //stand를 하여 딜러의 bust를 기대해본다.
                    }
                }
                return false;
            }
        }
    }
    else
    {   //simple player & dealer 의 경우
        if (SumofNum(hand) >= 17)
        {
            return true;    //stand 반환
        }
        else return false;  //hit 반환
    }
}
```

HitStatus()함수의 작동 방식은 17 미만의 값에서 hit하고 17 이상의 값에서 stand하는 플레이어2와 딜러의 작동방식을 기본으로 하지만 승률을 높이기 위해 예외처리 해주었고 그 작동방식에 있어 가장 중요한 부분은 좀 전에 정의한 CardCounting_for_decting_dealer_bust() 함수를 통해 반환된 딜러의 버스트 확률 CardCounting_for_decting_counting_player_bust() 함수를 통해 반환된 카운팅 플레이어의 버스트 확률을 이용하여 17 미만에서 hit을 하기 전에 카운팅 플레이어가 버스트할 확률을 통해 stand 할지 결정할 수 있고 또한 만약 이때 정한 stand하기 위한 버스트 확률 미만일 경우 다시 hit 하게 되는 이때 다시 카운팅 플레이어의 버스트 확률을 보고 버스트 확률이 0인지 보고 확실히 버스트 안 할 시 다시 한번 hit하게 한다. 또한 다음 예외처리로 만약 딜러가 무조건 hit을 해야하는 두 번째 카드가 2부터 5까지의 카드가 나오게 되면 CardCounting_for_decting_dealer_bust() 함수를 통해 반환된 딜러의 버스트 확률을 참고하여 stand하게하여 딜러의 버스트를 기대해 주는 방식으로 코딩하였다.

playgame() 함수

playgame() 함수는 블랙잭 게임의 메인역할을 하는 함수로 게임 실행을 담당한다. 코드가 긴 관계로 중요한 부분만 보면 다음과 같다.

```
for (int j = 0; j < game_count; j++)
{
    //입력한수만큼 진행
    //
    //80% 이상 사용했으면 Shuffle
    if (card_deck.size() < 32)
    {       //80%이상 사용되었다면
        for (int i = 0; i < 14; i++)
        {
            card_count[i] = 0;  //카운팅 초기화
        }
        Shuffle(&card_deck);    //Shuffle
    }

    // 딜러 ,플레이어 카드 초기화
    for (int i = 0; i < 17; i++)
    {
        simple_hand[i] = 0;
        counting_hand[i] = 0;
        dealer_hand[i] = 0;
    }
    // 플레이어 배팅 금액 설정
    simple_player_betting = Betting(card_count, false);
    counting_player_betting = Betting(card_count, true);
```

먼저 카드의 사용이 80%이상 사용되었으면 Shuffle함수를 통해 다시 한번 카드를 섞어주는 것을 알 수 있다.

```
temp = card_deck.back();
card_deck.pop_back();
dealer_hand[0] = temp;
first_card_of_dealer = temp;

temp = card_deck.back();
card_deck.pop_back();
counting_hand[0] = temp;
card_count[temp]++;

temp = card_deck.back();
card_deck.pop_back();
simple_hand[0] = temp;
card_count[temp]++;

temp = card_deck.back();
card_deck.pop_back();
dealer_hand[1] = temp;
second_card_of_dealer = temp;
card_count[temp]++;

temp = card_deck.back();
card_deck.pop_back();
counting_hand[1] = temp;
card_count[temp]++;

temp = card_deck.back();
card_deck.pop_back();
simple_hand[1] = temp;
card_count[temp]++;
```

플레이어 1,2와 딜러가 카드를 2장씩 받고 각 카드를 카운터에 저장하여 개수를 파악할 수 있게 한다.

```
// Card Counting player 게임 시작 -> 카드 카운팅 하여 hit or stand한다.
for (int i = 2; i < 17; i++)
{
    if (HitStatus(counting_hand, card_count, second_card_of_dealer, true) == false)
    {
        temp = card_deck.back();
        card_deck.pop_back();
        counting_hand[i] = temp;
        card_count[temp]++;
    }
    else
        break;  //stand
}
card_count[first_card_of_dealer]++; //딜러의 첫번째 패를 카운팅 함
first_card_of_dealer = 0;

// Simple player 게임 시작 -> 딜러 규칙대로 진행한다.
for (int i = 2; i < 17; i++) {

    if (HitStatus(simple_hand, card_count, second_card_of_dealer, false) == false)
    {
        temp = card_deck.back();
        card_deck.pop_back();
        simple_hand[i] = temp;
        card_count[temp]++;
    }
    else
        break;  //stand
}
// 딜러 게임 시작
for (int i = 2; i < 17; i++) {
    if (HitStatus(dealer_hand, card_count, second_card_of_dealer, false) == false)
    {
        temp = card_deck.back();
        card_deck.pop_back();
        dealer_hand[i] = temp;
        card_count[temp]++;
    }
    else
        break; //stand
}
```

플레이어 1, 2와 딜러가 각각의 게임을 실행하게 한다. 이때 HitStatus함수를 통해 Hit or Stand를 결정하고 stand가 되면 다음 사람의 게임이 실행된다.

```
    }

    // 플레이어들의 카드가 21보다 크다면? 딜러의 승으로 게임 끝
    if (SumofNum(counting_hand) > 21) {
        counting_p->budget -= counting_player_betting;  // 베팅금액만큼 잃는다.
        counting_p->rose += 1;  //rose count
        counting_bust_check = true;
    }
    if (SumofNum(simple_hand) > 21) {
        simple_p->budget -= simple_player_betting;
        simple_p->rose += 1;//rose count
        simple_bust_check = true;
    }

    // 딜러의 카드가 플레이어 보다 클때

    counting_card_sum = SumofNum(counting_hand);
    simple_card_sum = SumofNum(simple_hand);
    dlealer_card_sum = SumofNum_for_dealer(dealer_hand);
```

예외문을 사용하여 플레이어의 버스트를 파악하여 게임의 끝을 낼 수 있게 한다.

```
// 카운팅 플레이어 먼저
if (counting_bust_check == false)
{
    //카운팅 플레이어가 bust하지 않았을 경우
    if (dlealer_card_sum > counting_card_sum)
    {
        if (dlealer_card_sum > 21)
        {
            // 카운팅 플레이어 승
            counting_p->budget += counting_player_betting;  // 베팅금액 2배만큼 얻는다.
            counting_p->win += 1;   //win count
        }
        else {
            //카운팅 플레이어 짐
            counting_p->budget -= counting_player_betting;  // 베팅금액만큼 잃는다,
            counting_p->rose += 1;  //rose count

        }
    }
    else if (dlealer_card_sum == counting_card_sum)
    {
        //무승부
        counting_p->draw += 1;  //draw count
    }
    else
    {
        // 카운팅 블랙잭으로 플레이어 승
        if (counting_card_sum == 21)
        {
            counting_p->budget += 1.5*counting_player_betting;  // 베팅금액의 2.5배 만큼 얻는다.
            counting_p->win += 1;
        }
        else
        {
            counting_p->budget += counting_player_betting;  // 베팅금액만큼 얻는다.
            counting_p->win += 1;

        }

    }
}
```

```
    }
    if (simple_bust_check == false)
    {
        if (dlealer_card_sum > simple_card_sum)
        {
            if (dlealer_card_sum > 21)
            {

                simple_p->budget += simple_player_betting;
                simple_p->win += 1; //win count
            }
            else
            {

                simple_p->budget -= simple_player_betting;
                simple_p->rose += 1;//rose count


            }
        }
        else if (dlealer_card_sum == simple_card_sum)
        {
            simple_p->draw += 1;    //draw count
        }
        else
        {

            if (simple_card_sum == 21)
            {
                simple_p->budget += 1.5+simple_player_betting;
                simple_p->win += 1;
            }
            else
            {

                simple_p->budget += simple_player_betting;
                simple_p->win += 1;

            }
        }
    }
    counting_bust_check = false;
    simple_bust_check = false;
```

버스트하지 않고 통과하면 플레이어1와 플레이어2의 승, 패, 무승부를 기록한다.

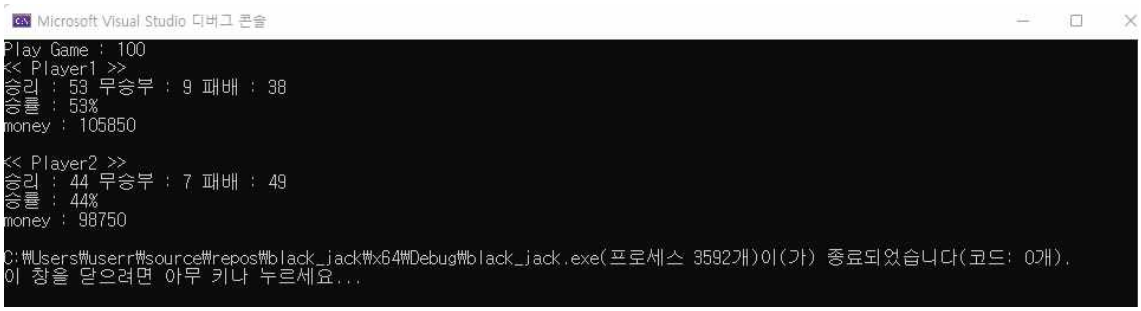
main() 함수

```
int main()
{

    int game_count = 0;     //게임 횟수
    cout << "Play Game : ";
    cin >> game_count;
    srand((unsigned int)time(NULL));

    struct player player_card_counting;
    player_card_counting.budget = 100000.0; //player의 예산 10만원으로 초기자본 부여
    player_card_counting.win = 0;//승리 횟수
    player_card_counting.draw = 0;//비긴 횟수
    player_card_counting.rose = 0;//진 횟수

    struct player player_simple;//player설정
    player_simple.budget = 100000.0;//player의 예산 10만원으로 초기자본 부여
    player_simple.win = 0;  //승리 횟수
    player_simple.draw = 0; //비긴 횟수
    player_simple.rose = 0; //진 횟수
    playgame(game_count, &player_simple, &player_card_counting);

    cout << "<< Player1 >>₩n"
        << "승리 : " << player_card_counting.win
        << " 무승부 : " << player_card_counting.draw
        << " 패배 : "<<player_card_counting.rose << endl
        << "승률 : " << 100 * ((double)player_card_counting.win / (double)game_count)
        << "%₩n"
        <<"money : " << player_card_counting.budget << endl;

    cout << endl;

    cout << "<< Player2 >>₩n"
        << "승리 : " << player_simple.win
        << " 무승부 : " << player_simple.draw
        << " 패배 : " << player_simple.rose << endl
        << "승률 : " << 100 * ((double)player_simple.win / (double)game_count)
        << "%₩n"
        << "money : " << player_simple.budget << endl;
    return 0;

}
```

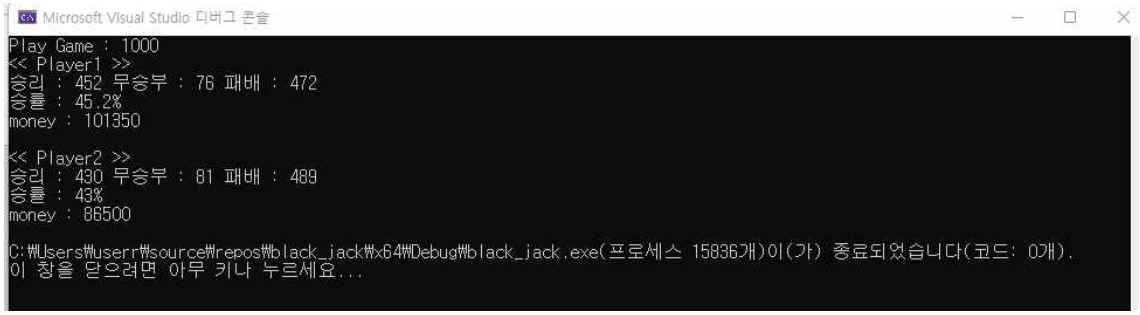
게임 실행 횟수를 입력받고 플레이어들의 구조체를 초기화 시킨다.

-SW 실행 결과

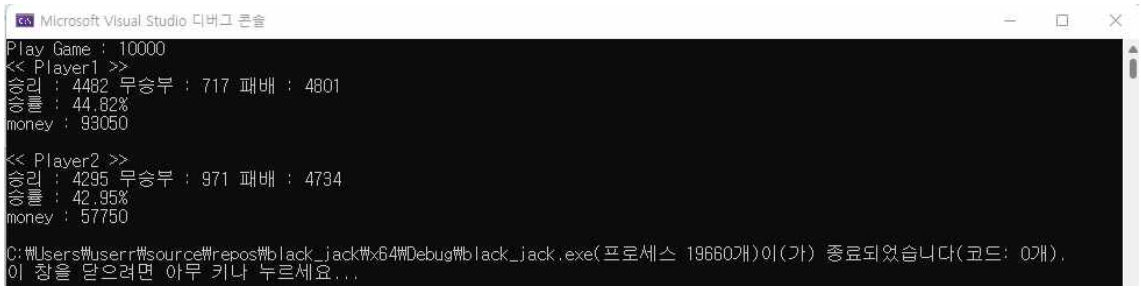
100번 실행.



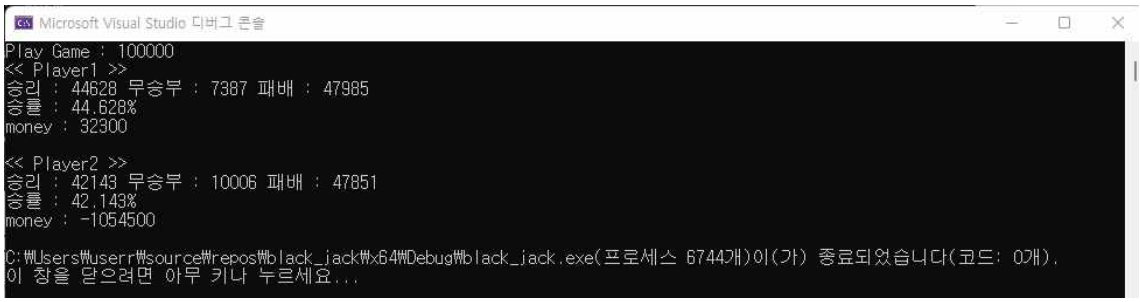
1000번 실행.



10000번 실행



100000번 실행



플레이어1이 카드카운팅을 통해 배팅과 hit or stand를 판단하여 정해진 규칙에 따라 플레이하는 플레이어2 보다 승리 횟수가 많고 돈은 잃었지만 플레이어2 보다 적은 돈을 잃었다.

-고찰

먼저 이번 프로그램을 설계하면서 딜러를 이길 수 있도록 카드 카운팅을 하고 Hit or stand를 판단하여 승률 50% 이상으로 설계를 하고 싶었지만 그렇지 못하였다. 하지만 HitStatus() 함수를 보다 효율적으로 코딩을 하면 이를 가능하게 한다고 생각한다.

이유는 다음과 같다.

HitStatus()함수의 카드 카운팅을 사용하는 플레이어를 위한 부분이다.

```
//card counting player의 경우
//카드 카운팅 사용
dlaer_bust_percentage = CardCounting_for_decting_dealer_bust(card_count, second_card_of_dealer);//딜러의 버스트 확률 구함.
counting_player_bust_percentage = CardCounting_for_decting_counting_player_bust(card_count, hand);// 카운팅 플레이어의 버스트 확률 구함.
if (SumofNum(hand) >= 17)
{
    return true;    //stand 반환
}
else
{
    //손에 있는 카드합이 17이하인 경우
    if (counting_player_bust_percentage > 40)
    {
        return true;    //플레이어가 버스트 할 확률이 50포함 크다면 stand
    }
    else
    {
        // 플레이어가 버스트 할 확률이 50%보다 작을때
        if (counting_player_bust_percentage == 0)
        {
            //플레이어가 버스트 할 확률이 0이면  담음카드에 모든걸 걸어본다.

            return false;   //hit한다.
        }
        if (2 <= second_card_of_dealer && second_card_of_dealer <= 5)
        {
            // A 가 첫번째로 나와도 무조건 hit를 해야 한다.
            // 이때 10 이상의 카드가 나올 확률이 50%가 넘는다면
            // 딜러가 스스로 bust할 확률이 높으므로 player는 stand하게 된다.
            if (dlaer_bust_percentage >= 40)
            {
                //cout << "ㅎㅁ" << endl;
                //딜러가 버스트 할 확률이 40%포함 크다면 stand
                return true;    //stand를 하여 딜러의 bust를 기대해본다.
            }
        }
        return false;
    }
}
```

먼저 이렇게 설계한 이유는 기존의 딜러의 게임 방식과 플레이어의 게임방식이 같을 때 확률은 반반이라고 생각했지만 딜러와 플레이어가 둘다 버스트일 경우 플레이어의 패가 됨으로 정확히는 40%센트 안팍의 확률을 보여주었다. 그래서 이 확률을 높여서 설계를 하여야 했기 때문에 기존의 딜러의 방식을 사용하면서 나만의 예외 구문을 설정해 주어 확률을 높이는 방식을 채택하였다. 그것이 위와 같은 형식이며 다음과 같다.

먼저 17이하에서 히트를 할 경우 그 때의 카운팅 플레이어의 버스트 확률이 40보다 크다면 버스트의 위험이 크게 있다 판단되어 stand를 하게 되었고 또한 이 구문을 실행하지 않고 else로 가게 된다면 이번엔 확실하게 카운팅 플레이어가 버스트가 되지 않을 확률인 0%일 경우 무조건 hit을 하게 만들어 주었다. 또한 이 구문을 마찬가지로 실행하지 않고 다음 if문을 실행할 때 딜러의 두 번째 카드를 보고 만약 2이상 5이하라면 첫 번째 카드가 무엇이든 상관없이 세 번째 카드를 뽑아야 하는 히트 상태에 있음을 이용 이때 만약 딜러의 버스트 확률 40포함 크다면 버스트 확률이 크다고 판단하여 딜러가 버스트 함을 기대하고 stand를 하게 하였다. 이후 이도 저도 아닌 애매한 경우 일 때 카드를 한 번 더 받아 다음 카드가 들어온 후 HitStatus를 재 실행하여 판단하는 것으로 설계하였다. 이것이 가장 최적의 방법은 아니라고 생각되며 다른 다양한 방법이 있다 생각 된다. 물론 프로그램을 설계할 때 이보다 더 좋은 예외 구문은 찾을 수 없었지만 다양한 방식으로 구문을 설계할 수 있음을 알게 되었다.